Drone-based Disaster Monitoring

By

Mithil Shah (2020A7PS0980P)

Dhruv Singh (2020B4A70969P)

Chirag Maheshwari (2020A7PS0983P)

CS F425 - Deep Learning

Under the guidance of Pratik Narang

BIRLA INSTITUTE OF TECHNOLOGY AND SCIENCE PILANI, PILANI CAMPUS

November 20, 2023

Drone-based Disaster Monitoring

Group No: F6

1. Introduction

This report discusses the implementation of deep learning models for drone-based disaster monitoring. The objective is to develop models for two primary tasks: Image Inpainting and Image Classification, focusing on disaster-related images.

2. Dataset Description

The dataset comprises images from five categories: Earthquake, Hurricane, Landslides, Flood, and Wildfire. It's divided into training and validation sets, with both original and corrupted images. The PairedDataset class handles dataset preparation, ensuring each category is equally represented.

TASK 1

Image Inpainting: Using Generative Adversarial Networks (GANs) to fill in corrupted parts of images

Let's walk through how a 256x256 RGB image is processed through each layer of the Generator and Discriminator of this cGAN architecture:

3. Generator

1. Input: The input is a 256x256 RGB image, meaning it has 3 channels.

2. First Convolutional Layer:
   - Layer: Conv2d(3, 64, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)
   - The image is convolved with 64 filters of size 4x4 with a stride of 2 and padding of 1. The output size is reduced to (256 + 2 * 1 - 4) / 2 + 1 = 128x128, resulting in 64 feature maps of size 128x128.

3. First Activation:
   - Layer: LeakyReLU(negative_slope=0.2, inplace=True)
   - Applies the LeakyReLU activation function to each of the 64 feature maps.

4. Second Convolutional Layer:
   - Layer: Conv2d(64, 128, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)
   - Further convolves each of the 64 feature maps with 128 filters of size 4x4, stride of 2, and padding of 1, halving the dimensions again to 64x64.

5. Second Batch Normalization and Activation:
   - Layers: BatchNorm2d and LeakyReLU as before
   - Normalizes the output of the second convolution and applies LeakyReLU, keeping the size the same.

6. Third Convolutional Layer:
   - Layer: Conv2d(128, 256, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)
   - Reduces the feature map size further to 32x32 with 256 channels.

7. Third Batch Normalization and Activation:
   - Again applies batch normalization and LeakyReLU activation.

8. First Transposed Convolutional Layer:
   - Layer: ConvTranspose2d(256, 128, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)
   - Upsamples the feature maps back to 64x64 with 128 channels.

9. Fourth Batch Normalization and Activation:
   - Applies batch normalization and a ReLU activation function.

10. Second Transposed Convolutional Layer:
   - Layer: ConvTranspose2d(128, 64, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)
   - Increases the size of the feature maps to 128x128 with 64 channels.

11. Fifth Batch Normalization and Activation:
   - Applies another batch normalization and ReLU activation.

12. Third Transposed Convolutional Layer:

- Layer: ConvTranspose2d(64, 3, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)
- The final layer up samples to the original image size of 256x256 and reduces the channels to 3 (RGB).

13. Tanh Activation:
    - Normalizes the output pixel values to the range [-1, 1].

4. Discriminator

1. Input: The input to the Discriminator is the concatenated real and generated images, resulting in a 6-channel input of size 256x256.
2. First Convolutional Layer:
    - Layer: Conv2d(6, 64, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)
    - Processes the 6-channel input and produces 64 feature maps of size 128x128.
3. First Activation:
    - Layer: LeakyReLU(negative_slope=0.2, inplace=True)
    - Applies LeakyReLU activation to the feature maps.
4. Second Convolutional Layer:
    - Layer: Conv2d(64, 128, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)
    - Produces 128 feature maps of size 64x64.
5. Second Batch Normalization and Activation:
    - Layers: BatchNorm2d and LeakyReLU
    - Normalizes and activates the feature maps.
6. Third Convolutional Layer:
    - Layer: Conv2d(128, 256, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)
    - Produces 256 feature maps of size 32x32.
7. Third Batch Normalization and Activation:
    - Continues the pattern of batch normalization followed by LeakyReLU activation.
8. Fourth Convolutional Layer:
    - Layer: Conv2d(256, 1, kernel_size=(4, 4), stride=(1, 1), bias=False)
    - Reduces the feature maps to a single channel of size 29x29 (since no padding is used and stride is 1).
9. Sigmoid Activation:
    - Layer: Sigmoid()

- Transforms the single channel into a probability map, where each value represents how likely a corresponding patch in the input image is real.

The output of the Discriminator is typically further processed, such as by averaging, to give a single probability score indicating whether the input image is real or fake.

Generator: Consists of alternating convolutional and transposed convolutional layers with LeakyReLU activation and Batch Normalization. The final layer uses a Tanh activation function. This design aims to generate high-fidelity images.
Discriminator: Built with convolutional layers and LeakyReLU activation, ending with a Sigmoid function to classify images as real or fake.
Initialization: Xavier initialization is used to optimize the training stability.
CNN Architecture for Image Classification
Model: Utilizes a pre-trained ResNet50 model with a custom final layer to classify five disaster categories.
Layer Modification: The last fully connected layer is adjusted for the specific task, and certain layers are frozen to retain pre-trained features.

TASK 2

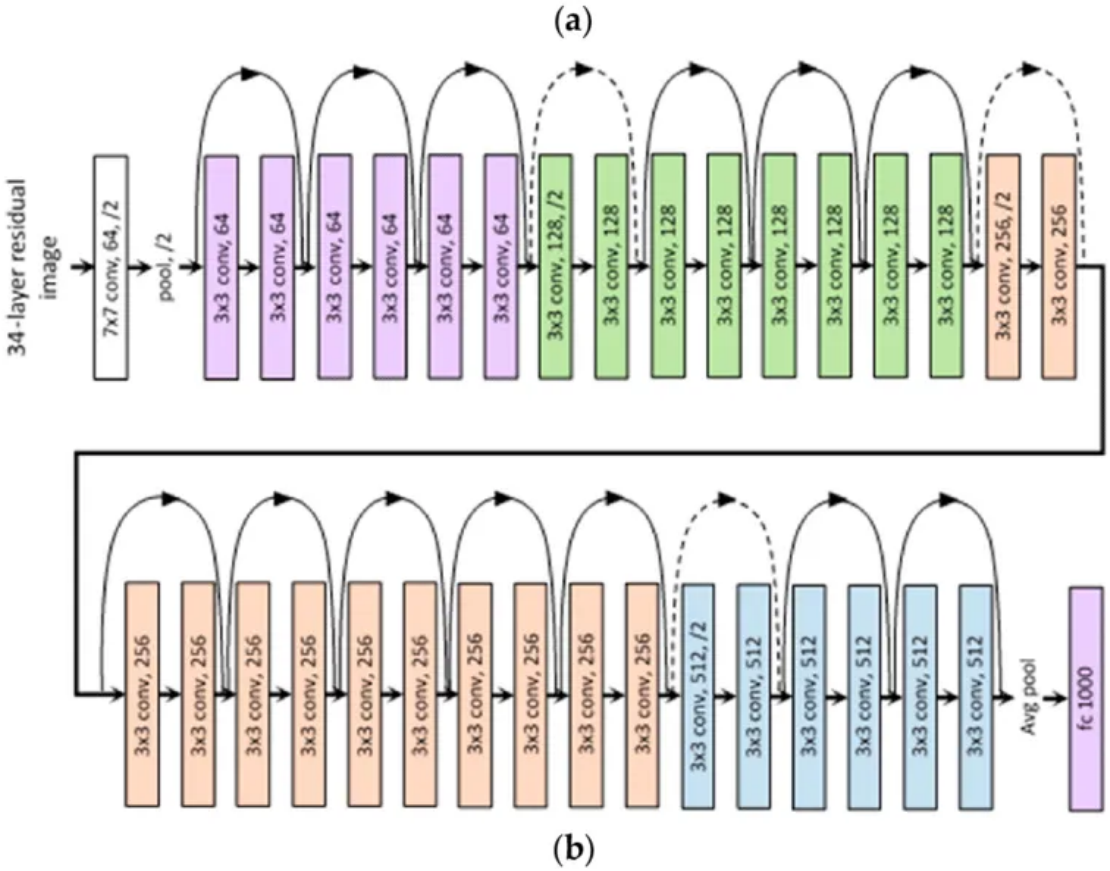
Image Classification: Classifying images into five categories

We have done the classification by utilizing Transfer Learning by using a pre-trained Convolutional Neural Network (CNN), specifically a modified ResNet50 architecture. ResNet-50 is part of the ResNet (Residual Network) family of models, which was a breakthrough in the field of deep learning when it was introduced. The "50" in its name refers to the number of layers it has, which amounts to 50.

5. Structure of ResNet-50

ResNet-50 is a convolutional neural network (CNN) with 50 layers deep. The main components of ResNet-50 are:
1. **Convolutional Layers:** These are the layers that perform convolution operations, which extract features from the input images by sliding filters over the image and producing a feature map.
2. **Batch Normalization:** This is used after each convolution operation to stabilize learning by normalizing the layer's inputs by mean and variance.
3. **Activation Function:** ResNet-50 uses ReLU (Rectified Linear Unit) as the activation function, which introduces non-linearity into the network.

4. **Residual Blocks**: These are the fundamental units of ResNet. Each block contains a shortcut connection that skips one or more layers. In ResNet-50, each block contains 3 layers, and these blocks are known as "bottleneck" blocks.
5. **Pooling Layers:** They are used to reduce the spatial dimensions of the feature maps.
6. Fully Connected Layer: At the end of the network, a fully connected layer is used for classification.
7. **Softmax Layer**: This layer outputs the probabilities for each class.



Why ResNet-50 is Effective:

The key innovation of ResNet is the introduction of "shortcut connections" or "skip connections," which allow the gradient to bypass certain layers during backpropagation. This addresses the problem of vanishing gradients, which is a common issue when training deeper networks. Here are some reasons why it is effective:

1. **Solves Vanishing Gradient Problem:** The skip connections allow gradients to flow through the network without being diminished, making it possible to train very deep networks.
2. **Ease of Training:** Residual blocks make it easier for the network to learn identity mappings, which means layers can effectively learn to preserve the signal from the input of the block to its output.
3. **Feature Reuse**: The architecture allows the model to reuse features from earlier layers, which makes the network more parameter-efficient.
4. **Reduced Training Error:** With residual blocks, training error decreases as the depth of the network increases, which was not the case with traditional deep networks.
5. **Versatility**: ResNet-50 has shown remarkable performance on a variety of tasks beyond image classification, like object detection and segmentation.
6. **Pre-Trained Models Availability**: Due to its popularity, there are pre-trained ResNet-50 models available that have been trained on large datasets like ImageNet, making it a strong starting point for transfer learning.
In summary, ResNet-50's architecture allows it to be deep enough to capture complex features while avoiding the pitfalls that typically come with increased depth in neural networks. This balance of depth and stability during training is what makes ResNet-50 and its variants so powerful in practice.

6. Training Process

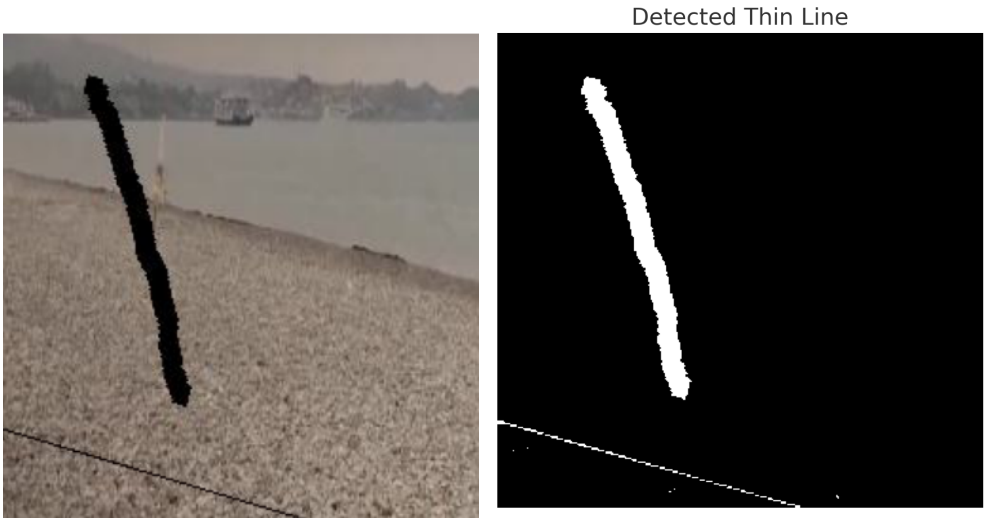
GAN Training: Involves alternate training of the Discriminator and Generator using BCELoss.

- Normalization of image inputs was done on the images for improve gradient flow, prevent activation function saturation and faster convergence.
- Xavier initialization was used to initialize the weights of the cGAN.
- Adam optimizer with specific learning rates is employed.
- LR Scheduler was used to dynamically change the learning rate.

CNN Training: The ResNet50 model is fine-tuned using CrossEntropyLoss, focusing on the last fully connected layer.

7. Innovation

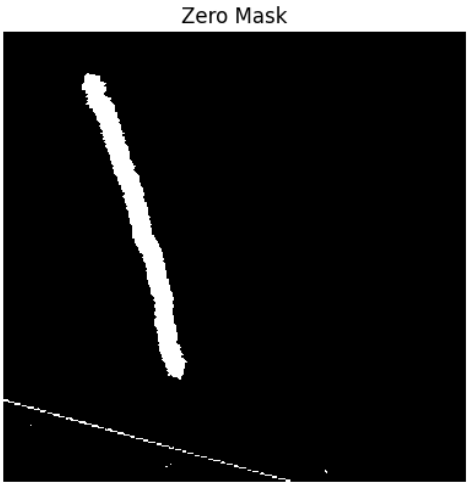
1) Detection of Corrupted Parts in the Image:
    - Using the CV2 model imported from Open CV, we could easily see the corrupted part, as shown below



However, instead of using an online library to do this, we chose to go with our own implementation to detect the corrupted part, as shown in the next point. Both gave similar results so we just chose to go with a simpler version (our own version).

- Find the pixels from which there is a dark region; this could be achieved by the code shown below. The code tries to find all the pixels from which the RBG values are strictly between 0 and 0.07.

```
mask = (corrupted_imgs >= 0) & (corrupted_imgs <= 0.07)
zero_mask = mask.all(dim=1, keepdim=True)
```



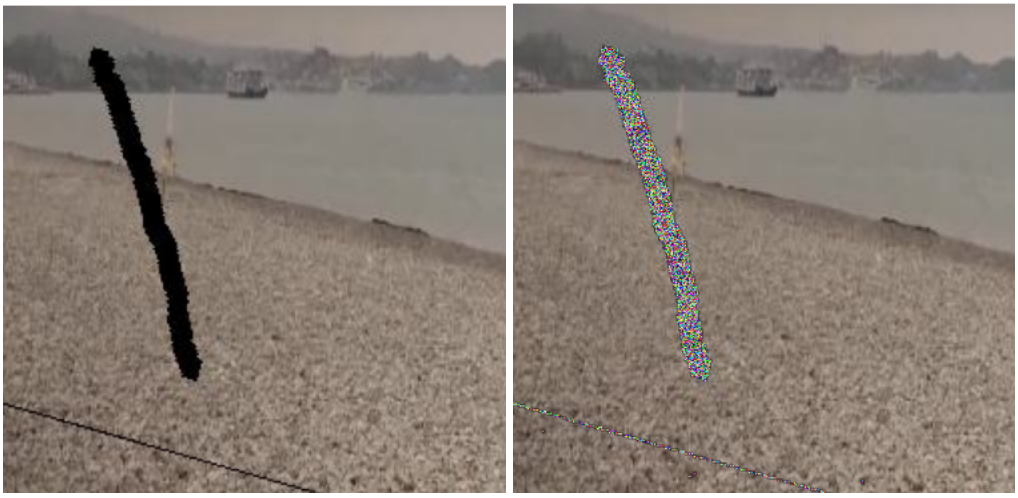
2) Initialization of the model weights

- As learned in DL class, instead of normal initialization, we used Xavier initialization for our model, as shown below.

```
def weights_init_xavier(m):
    if isinstance(m, nn.Conv2d) or isinstance(m, nn.Linear) or isinstance(m, nn.ConvTranspose2d):
        init.xavier_uniform_(m.weight.data)
        if m.bias is not None:
            init.constant_(m.bias.data, 0.0)

G.apply(weights_init_xavier)
D.apply(weights_init_xavier)
```

3) Image sent to Generator

- Instead of sending the corrupted image to the generator, we thought about pre-processing the generated image; this pre-processing the image involved replacing the black part (the corrupted part) with random noise. Since the black part is at one extreme of the spectrum, we thought, why not replace all the black portion with random noise, as shown below:



4) Preprocessing the Image

- We used standard normal distribution to convert the pixel values between [-1, 1] through the following code:

```
transform_normalized = transforms.Compose([
    transforms.Resize((256, 256)),
    transforms.ToTensor(),
    transforms.Normalize([0.5 for _ in range(3)], [0.5 for _ in range(3)]),
])
```

The above code ensures that the image is 256 px in both width and height, and the pixel values are then converted from [0,1] to [-1,1] using the standard normal distribution.

8. Hyperparameters and Model Parameters

Optimizers:

Adam Optimizer: The Adam optimizer is a popular choice for deep learning models as it combines the best properties of the AdaGrad and RMSProp algorithms to handle sparse gradients on noisy problems.

Learning Rates:

Learning Rate for G and D (0.0002): This is a relatively low learning rate, which is typical for GANs as it can help stabilize training and avoid the generator and discriminator oscillating or diverging during training.

Betas:

Betas (0.5, 0.999): These values adjust the decay rates of the moving averages of the gradient and its square, which are parameters of the Adam optimizer. A beta1 value of 0.5 means the optimizer is putting more weight on the more recent gradients, which can help with the stability of GAN training.
Number of parameters in the GAN and CNN models.

```
GAN
Total number of parameters in the Generator: 1318016
Total number of parameters in the Discriminator: 666368

Transfer Learning using CNN
Total parameters: 23518277
Learnable parameters: 10245
Unlearnable parameters: 23508032
```

9. Performance Metrics

Image Inpainting: SSIM and PSNR metrics are used to evaluate the quality of inpainted images are shown below:

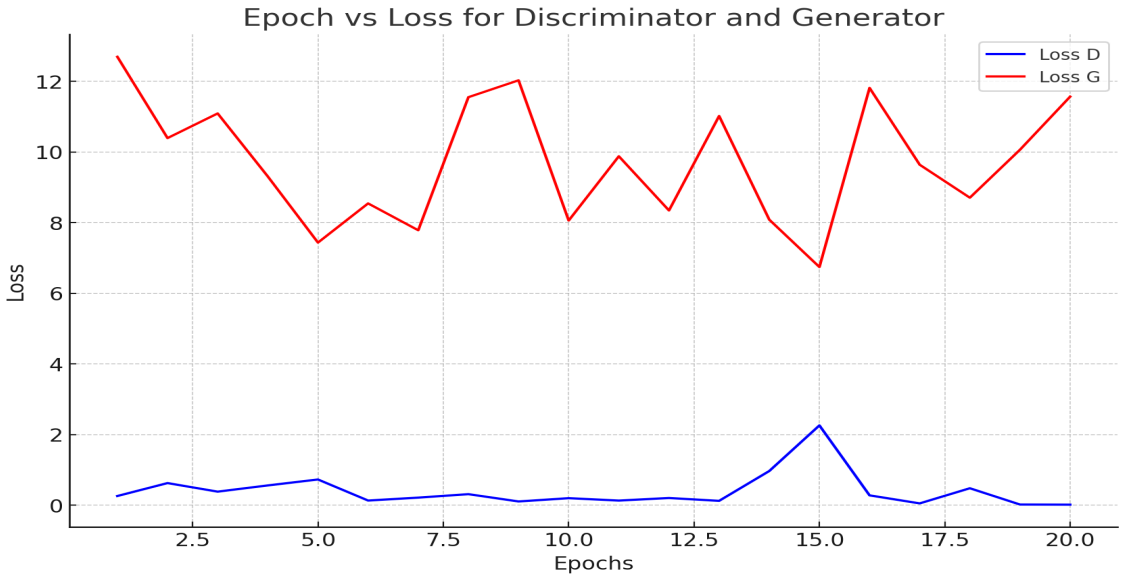
```
Mean SSIM: 0.9268
Mean PSNR: 29.4028


SSIM max :0.9994644884043428 min: 0.6568866602293213
PSNR max :52.09323911021721 min: 24.10109427004449
```

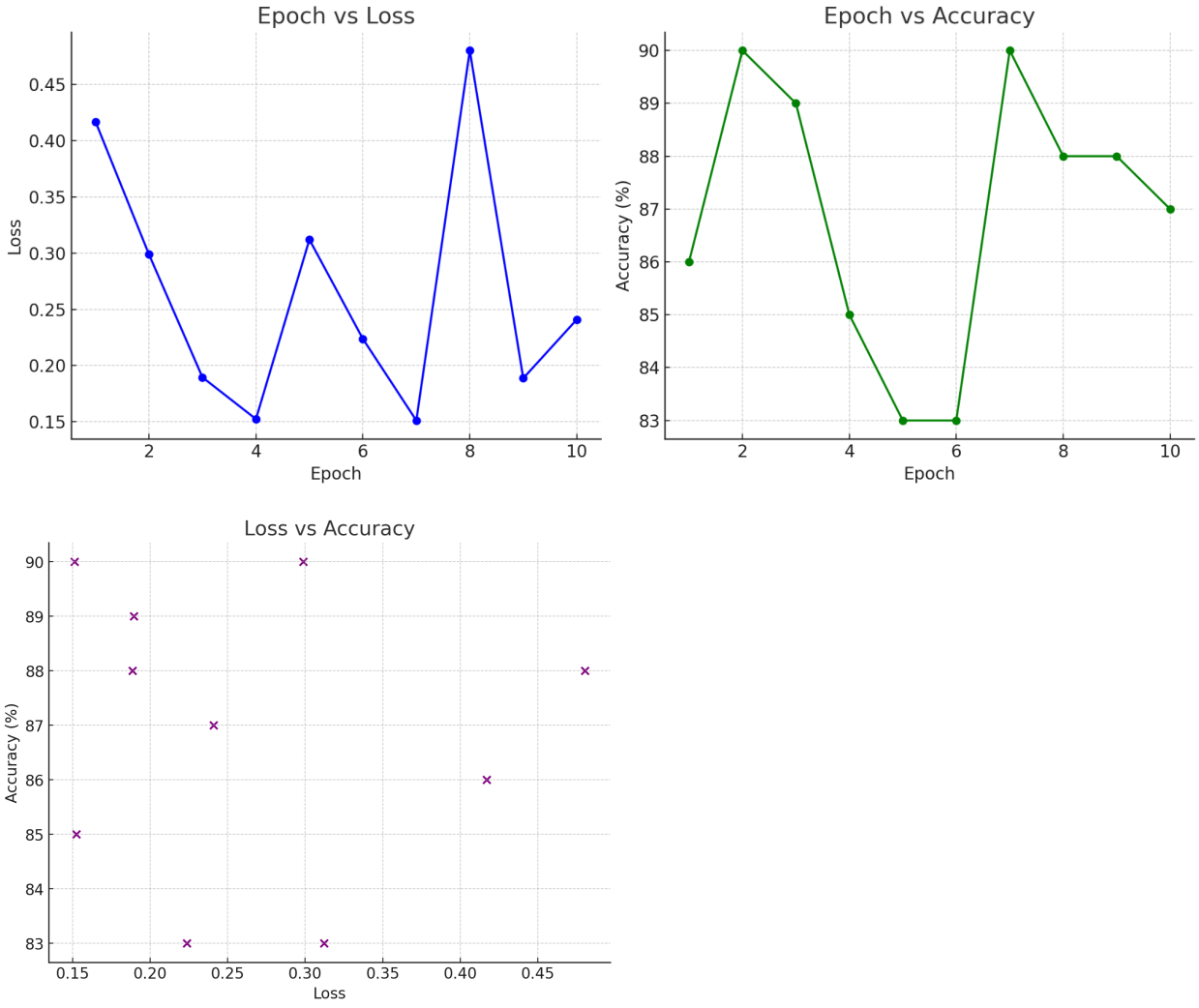
Image Classification: Accuracy, precision, and recall metrics assess the model's performance are shown below:

```
Class: Earthquake
Accuracy: 0.85
Precision: 0.87
Recall: 0.85
------------------------------
Class: Hurricane
Accuracy: 0.84
Precision: 0.76
Recall: 0.84
------------------------------
Class: Landslide
Accuracy: 0.98
Precision: 0.81
Recall: 0.98
------------------------------
Class: Flood
Accuracy: 0.81
Precision: 0.81
Recall: 0.81
------------------------------
Class: Wildfire
Accuracy: 0.85
Precision: 0.99
Recall: 0.85
------------------------------
OverAll
Accuracy: 0.85
Precision: 0.86
Recall: 0.85
```

10. Loss-Accuracy Curves

Include graphical representations of the loss and accuracy trends over the training and validation phases for both tasks.





11. Training and Testing Time

Report the time taken for training and testing both the GAN and CNN models.

The following are the time taken to complete training and testing:

Training:

- GAN: 19 min, 58 sec
- Time Taken for Transfer Learning using CNN: 1 minute, 42 seconds

Testing:

- GAN: 7 sec
- Time Taken for Transfer Learning using CNN: 0.2 sec

The above-mentioned times are calculated on a GPU machine. On the CPU machine, the timing can vary anywhere from 10 times the above values to 50 times.

12. Results (GAN)

13. Conclusion

The project effectively combines Image Inpainting and Image Classification for drone-based disaster monitoring using advanced deep learning techniques. It employs a conditional Generative Adversarial Network (cGAN) for Image Inpainting, restoring integrity to corrupted disaster-related images. For Image Classification, it utilizes a modified ResNet-50 architecture, leveraging transfer learning to efficiently categorize images into five disaster types. This blend of GANs and ResNet-50 demonstrates innovative approaches to real-world challenges in disaster analysis, showcasing the potential of deep learning in critical applications like disaster management and response.